HeavyGo — Smart On-Demand Transport System

Project Documentation Template

Prepared by: HeavyGo Team

November 30, 2025

Contents

Executive Summary

HeavyGo is an on-demand transport platform that connects clients who need items transported with drivers having suitable vehicles. Pricing and vehicle selection are based on item **size/volume**, weight and distance. The system is built with ASP.NET Core MVC, Entity Framework Core, ASP.NET Identity and SignalR for real-time features.

1 Project Overview

1.1 Objective

Build a scalable and user-friendly transport-ordering web application where:

- Clients create transport orders specifying pickup/dropoff and item size/volume.
- Drivers view nearby orders, propose or accept prices and complete deliveries.
- The system supports authentication, payments and real-time location updates.

1.2 Key Capabilities

1. Client order creation (pickup, dropoff, item dimensions).
2. Driver discovery of nearby orders (Haversine distance).
3. Request/acceptance workflow (DriverOrderRequest).
4. Payment record creation and payment history.
5. Role-based access (Client, Driver, Admin).
6. Optional: Live driver tracking via SignalR.

2 Architecture & Design Patterns

2.1 Architecture Layers

Presentation Layer: ASP.NET Core MVC controllers and Razor views.

Business Logic Layer: Services (or controllers) containing business rules (pricing logic, matching).

Data Access Layer: EF Core DbContext (optionally with Repository/Unit of Work).

2.2 Design Patterns to Use

- MVC (Model-View-Controller)
- Dependency Injection (built into ASP.NET Core)
- (Optional) Repository Pattern + Unit of Work for clearer separation of data access.
- SignalR for real-time messaging.

3 Data Model (High-level)

- **ApplicationUser** – extends Identity user; stores basic location coordinates.
- **Order** – OrderId, ClientId, PickupLat/Long, DropoffLat/Long, ItemVolume, ItemWeight, TotalPrice, Status, CreatedAt.
- **DriverOrderRequest** – RequestId, OrderId, DriverId, Status (Pending/Accepted/Rejected), ProposedPrice, RespondedAt.
- **Payment** – PaymentId, OrderId, Amount, PaymentMethod, PaidAt, TransactionRef.
- **Review** – ReviewId, OrderId, Rating, Comment, CreatedAt.

4 Week-by-Week Development Plan (Template)

Week 1 — Core Setup

- Configure solution and projects: `HeavyGo.Web` (MVC), `HeavyGo.Data` (EF models), optionally `HeavyGo.Services`.
- Setup EF Core DbContext and Identity (roles: Client, Driver, Admin).
- Implement basic OrdersController: Create, List, Details.
- Seed roles and test users (optional).

Week 2 — Driver Workflow & Distance Filtering

- Implement Haversine distance helper and filter orders by proximity to driver location.
- DriverOrderRequests flow: create request, accept/reject.
- Protect pages with role-based authorization attributes.

Week 3 — Payments & Reviews

- Implement PaymentsController and store payment records.
- Integrate (or stub) a payment gateway (Stripe recommended for production).
- Implement Reviews for completed orders.
- Add SignalR stub for real-time location updates.

Week 4 — Polish, Logging, Tests

- Improve UI with Bootstrap; add client and driver dashboards.
- Add structured logging (Serilog recommended) and health checks.
- Run end-to-end tests and prepare deployment steps.

5 Important Controllers & Code Pointers

Program.cs: Register services, Identity, DbContext, SignalR hub endpoints and default routing.

OrdersController: CRUD for orders. Add fields: ItemVolume, ItemWeight and validation.

HomeController: Driver home view showing nearby orders; implement AcceptOrder with concurrency safety.

DriverOrderRequestsController: Manage driver responses.

PaymentsController: Create and list payments; link to order.

6 Pricing Logic (Recommendation)

Use a small pricing service to compute suggested price:

1. Base price per kilometer (distance via Haversine).
2. Volume/weight multiplier (e.g., price += baseRate * volumeFactor).

3. Vehicle type surcharge (truck, van, small truck).

Implement as a testable C# service:

```
IPricingService.GetSuggestedPrice(Order order, Location pickup, Location dropoff)
```

7 Security & Roles

- Use ASP.NET Identity: create ‘Client‘, ‘Driver‘, and ‘Admin‘ roles at startup (seed).
- Use ‘[Authorize(Roles="Driver")]‘ and ‘[Authorize(Roles="Client")]‘ where appropriate.
- Ensure server-side checks: a client can only edit/view their own orders; a driver can only act on their requests.

8 Concurrency & Data Integrity

When accepting an order, use a database transaction or optimistic concurrency to ensure only one driver can accept an order:

- Add a column ‘Order.Status‘ (Available/Assigned/Completed).
- Set ‘Order.Status = Assigned‘ inside a transaction after verifying it was ‘Available‘.

9 Known Limitations & Future Work

- Add explicit `ItemVolume`/`RequiredVehicleType` to Orders (if missing).
- Integrate a production payment gateway (Stripe) with webhook verification.
- Map UI integration (Google Maps / Mapbox) and SignalR-based live tracking.
- Move business logic into dedicated services for easier testing.

10 Deployment Notes

- Use environment variables for connection strings and secrets (do not store in plaintext).
- Configure migrations on CI/CD or apply at startup with caution.
- Set up a logging sink (e.g., Seq/Elasticsearch or Serilog file sink).